IBM ILOG Views

Gadgets V5.3

Tutorial

June 2009

© Copyright International Business Machines Corporation 1987, 2009.
US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

IBM, the IBM logo, ibm.com, Websphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Notices

For further information see <*installdir*>/license/notices.txt in the installed product.

Table of Contents

Preface — **About This Tutorial** 3

What You Need to Know 3

Notation 3

Typographic Conventions 3

Naming Conventions 4

Tutorial 1 — **The ViewFile Application: Building a Simple File Browser** 3

Step 1: Building the Browser Window 4

Choosing the Right Gadgets for your Application 4

Choosing the Container 5

Implementing the Main Window 6

Creating the Application 8

Previewing the Application 9

Step 2: Browsing Files 10

Creating the File Browser 10

Creating the Gadget Items 18

Instantiating the File Viewer 19

Previewing the Application 20

Step 3: Adding Docking Bars 20

Preparing the Main Window for Docking 21
Creating the Docking Toolbar 22
Creating the Docking Menu Bar 24
Integrating Changes into the Application 25
Previewing the Application 25
Step 4: Adding View Frames 26
Choosing the Desktop View 26
Creating View Frames 29
Adding New Menus and Items to View Frames 30
Previewing the Application 33

Tutorial 2 Customizing Gadgets 35

Step 1: Extending a Gadget by Changing its Graphical Appearance 35
Creating a Subclass of an Existing Gadget 36
Adding an API to the New Gadget Class 38
Modifying How the New Gadget is Drawn 41
Testing the New Gadget Class 43
Step 2: Extending a Gadget by Changing its Behavior 44
Choosing the Right Way to Modify the Behavior of an Existing Gadget 45
Creating a Generic Interactor 45
Creating a Dedicated Interactor 47
Testing the New Interactor 49
Step 3: Creating a Composite Gadget 50
Creating a Gadget Composed of Other Gadgets 50
Handling Keyboard Focus in a Gadget 53
Handling Events in a Gadget 55
Adding Callbacks to a Gadget 56
Testing the Composite Gadget 57

Index 59

About This Tutorial

This tutorial explains how to build a simple file browser and how to customize gadgets.

What You Need to Know

This manual assumes that you are familiar with the PC or UNIX® environment in which you are going to use IBM® ILOG® Views, including its particular windowing system. Since IBM ILOG Views is written for C++ developers, the documentation also assumes that you can write C++ code and that you are familiar with your C++ development environment so as to manipulate files and directories, use a text editor, and compile and run C++ programs.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- Code extracts and file names are written in `courier` typeface.
- Entries to be made by the user are written in `courier`.
- Some words appear in *italics* when seen for the first time.

Naming Conventions

Throughout this manual, the following naming conventions apply to the API.

- The names of types, classes, functions, and macros defined in the library begin with `Ilv`.
- The names of classes as well as global functions are written as concatenated words with each initial letter capitalized.

  class `IlvDrawingView`;

- The names of virtual and regular methods begin with a lowercase letter; the names of static methods start with an uppercase letter. For example:

```
virtual IlvClassInfo* getClassInfo() const;

static IlvClassInfo* ClassInfo*() const;
```

CHAPTER

1

The ViewFile Application: Building a Simple File Browser

This tutorial shows you how to create the ViewFile application using the IBM® ILOG® Views Gadgets API. The ViewFile application is a small graphical user interface for browsing through a list of files stored on a hard disk.

> ***Note:*** *You can also build the same kind of application using IBM ILOG Views Studio, the IBM ILOG Views GUI editor.*

As you progress through this tutorial, you will learn how to combine many different IBM ILOG Views gadgets, such as the tree gadget, the sheet, and the toolbar to produce very graphical and highly intuitive applications. Your final application will include sophisticated features such as docking toolbars and multiple frames.

This tutorial has four steps:

- *Step 1: Building the Browser Window*
- *Step 2: Browsing Files*
- *Step 3: Adding Docking Bars*
- *Step 4: Adding View Frames*

When completed, your application should look like this:

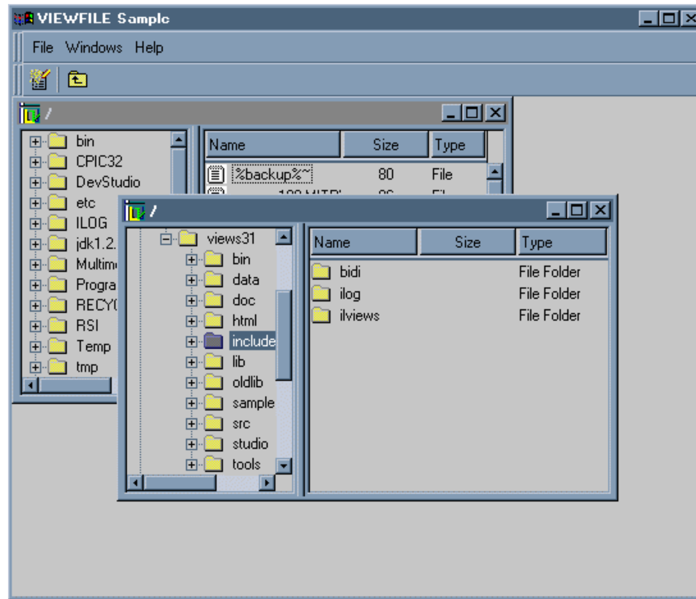


Figure 1.1 The Final ViewFile Application

Step 1: Building the Browser Window

This first step explains how to create the window that will display the file hierarchy.

It shows you how to perform the following tasks:

- *Choosing the Right Gadgets for your Application*
- *Choosing the Container*
- *Creating the Application*
- *Previewing the Application*

Choosing the Right Gadgets for your Application

The application window is composed of two parts:

- The left-hand side of the window displays the folders on the hard disk as a tree structure.
- The right-hand side of the window displays the files contained in the folder open on the left side.

In IBM® ILOG® Views, hierarchical lists of items are represented by the class IlvTreeGadget. This class will be used to build the left-hand side of the window. To represent the right-hand side of the window, the IlvSheet will be used. Instances of this class can display different types of information, thus allowing you to modify the representation of data without changing the type of the object that displays it.

Because the set of gadgets provided by IBM ILOG Views is large, you might have difficulty finding the right gadgets for your application. To get an overview of the main properties of IBM ILOG Views Gadgets, see the chapter Introducing IBM ILOG Views Gadgets. Each gadget is described in detail in its related section in the IBM ILOG Views *Gadgets User's Manual*.

If you do not find a gadget that suits your needs, you can subclass an existing gadget to modify its look or its behavior, or both.

Choosing the Container

The IlvTreeGadget and the IlvSheet objects must be added to a container that will display them. Although there are several types of containers you can choose from, only two are appropriate for the ViewFile application—IlvGadgetContainer and IlvPanedContainer.

IlvGadgetContainer is a subclass of IlvContainer that accommodates gadgets. It handles the keyboard focus and attachments. For information on these two features, see Gadgets Main Properties.

IlvPanedContainer is a subclass of IlvGadgetContainer. Unlike gadget containers, paned containers do not handle graphic objects. Instead, they handle special objects of the class IlvPane. Depending on the orientation of the paned container, which can be horizontal or vertical, panes can be arranged from left to right or from top to bottom. IlvPane has two predefined subclasses, one of which is the IlvGraphicPane class that accommodates graphic objects. Panes are described in length in Panes.

IlvPanedContainer is more appropriate for this application because it allows you to create a window composed of two adjacent panes having the same height—one for holding the tree gadget and one for the sheet. Also, the complex attachment model provided by IlvGadgetContainer is not needed. However, both types of containers can be combined in the same application.

For example, you can use gadget containers for building dialog boxes and complex graphic panels and therefore benefit from the elaborate attachment model they feature. Then you can encapsulate them into panes at the global application level.

Implementing the Main Window

The main window is implemented with the `FileViewerWindow` class, a subclass of `IlvPanedContainer`. This class is declared in the file `viewerw.h` and defined in the file `viewerw.cpp`.

Creating Panes

The class `FileViewerWindow` automatically creates two graphic panes: one for encapsulating the tree and the other one for encapsulating the sheet. Both panes are created

in the `FileViewerWindow::initLayout` member function, shown here. (This member function is called from the `FileViewerWindow` constructor):

```
void
FileViewerWindow::initLayout(const IlvRect& size)
{
    // Create the tree gadget in which the folder hierarchy will
    // be displayed.
    IlvTreeGadget* tree =
        new IlvTreeGadget(getDisplay(),
                          IlvRect(0,
                          0,
                          IlvMax((IlvDim)100,
                                 (IlvDim)(size.w()/3)),
                          size.h()));
    // Encapsulate the tree gadget into a graphic pane and name it
    // DirectoryHierarchy.
    IlvGraphicPane* treePane = new IlvGraphicPane("DirectoryHierarchy", tree);
    // The pane is set to resizable. It is possible to
    // resize it using a splitter interactively.
    treePane->setResizeMode(IlvPane::Resizable);
    // Set also a minimum size on the horizontal direction.
    treePane->setMinimumSize(IlvHorizontal, 100);
    // Add the pane to the container.
    addPane(treePane);

    // Create the sheet.
    IlvSheet* sheet =
        new IlvSheet(getDisplay(),
                     IlvRect(0, 0, size.w(), size.h()),
                     1,
                     1,
                     size.w()/4,
                     25,
                     2,
                     IlvFalse,
                     IlvFalse);
    // Encapsulate it into a graphic pane giving FileList as name.
    IlvGraphicPane* listPane = new IlvGraphicPane("FileList", sheet);
    // The sheet is elastic: when the container will be resized, only the
    // sheet will be resized (because the tree is only resizable, not elastic).
    listPane->setResizeMode(IlvPane::Elastic);
    // Add the pane to the container.
    addPane(listPane);
    // Update the container.
    updatePanes();
}
```

The `initLayout` member function first creates the tree gadget. Then, it encapsulates it into a graphic pane and adds it to the paned container. Similarly, it creates the sheet, encapsulates it into a graphic pane, and adds it to the paned container.

> ***Note:*** *The sheet has only one row and one column, because at this stage you do not know what the sheet will look like when the browsed files are displayed. Note also that the width of the sheet specified by the parameter* `(size.w(),4)` *is meaningless because the sheet is elastic and occupies the space left by the tree.*

The member function `IlvPanedContainer::updatePanes` is called to make the addition of the two graphic panes effective. For further information, see the section "Modifying the Layout of a Paned Container" in Panes.

> ***Note:*** *Calling the member function* `IlvPanedContainer::updatePanes` *inserts a slider pane in between the tree and the sheet panes automatically because both are resizable.*

Creating the Application

The class for displaying the files hierarchy is now complete and can be instantiated. To instantiate it, `FileViewerApplication`, a subclass of `IlvApplication`, is used. `IlvApplication` represents a basic application made up of a set of panels. It is used by IBM® ILOG® Views Studio for generating the code of an application.

The `FileViewerApplication` class is declared in the file `viewfile.h` and defined in the file `viewfile.cpp`.

Creating the Main Window

The `IlvApplication` class contains the `makePanels` virtual member function, which is invoked at the beginning of the program to create the application panels.

```
void
FileViewerApplication::makePanels()
{
    // Initialize the main window.
    initMainWindow();
    // Show it.
    getMainWindow()->show();
}
```

The member function `FileViewerApplication::initMainWindow` creates an instance of `FileViewerWindow`:

```
void
FileViewerApplication::initMainWindow()
{
    // Create the main window.
    IlvRect rect(0, 0, 500, 300);
    IlvContainer* mainWindow = createMainWindow(rect);
    // Name it to be able to retrieve it.
    mainWindow->setName(getName());
    // Quit the application when the user asks for termination.
    mainWindow->setDestroyCallback(IlvAppExit);
    // Add the panel to the application.
    addPanel(mainWindow);
}
```

The member function `createMainWindow` returns an instance of the class `FileViewerWindow`.

```
IlvContainer*
FileViewerApplication::createMainWindow(const IlvRect& rect) const
{
    return new
    FileViewerWindow(getDisplay(),getName(),getName(),rect,IlvFalse);
}
```

Instantiating the Application

The `FileViewerApplication` class is instantiated in the main entry point:

```
int main(int argc, char* argv[])
{
    IlvSetLocale();
    FileViewerApplication* appli =
        new FileViewerApplication("VIEWFILE Sample", 0, argc, argv);
    if (!appli->getDisplay())
        return -1;
    appli->run();
    return 0;
}
```

> ***Note:*** *The call to* `IlvSetLocale` *simply tells IBM ILOG Views to use the current locale. You must call this function if you want your application to be localized. For information, see "Creating a Program to Run in a Localized Environment" in Internationalization in IBM ILOG Views.*

Previewing the Application

This is the end of Step 1. Your application window should look like this:

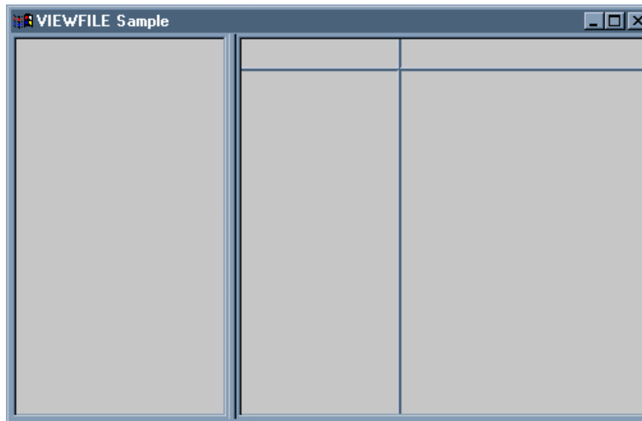


Figure 1.2 *The ViewFile Application After Step 1*

Step 2: Browsing Files

In *Step 1: Building the Browser Window* on page 4 you saw how to create the graphic objects for displaying a hierarchical list of files. This second step shows how to define and instantiate the file browser, that is, the object used to fill the graphic objects (the tree and the sheet). The class used to browse the hard disk is `FileViewer`. It is declared in the file `viewer.h` and defined in the file `viewer.cpp`.

This step shows you how to perform the following tasks:

- *Creating the File Browser*
- *Creating the Gadget Items*
- *Instantiating the File Viewer*
- *Previewing the Application*

> ***Note:*** *The classes* `IlPathName` *and* `IlString` *seen in the code for this step are used to make the code easier to read.*

Creating the File Browser

The `FileViewer` class is long and complex. This section explains its main parts:

- *The Constructor*
- *Initializing the Tree*

- *Initializing the Sheet*
- *Scanning Directories*
- *Updating the Sheet*

The Constructor

The `FileViewer` constructor requires an `IlvTreeGadget` and an `IlvSheet` as its parameters. Both objects are initialized by the `initObjects` member function:

```
FileViewer::FileViewer(IlvTreeGadget* tree, IlvSheet* sheet)
: _tree(tree), _sheet(sheet)
{
    // Initialize the gadgets.
    initObjects();
}
```

Initializing the Tree

The tree object is initialized with the `initTree` member function:

```
void
FileViewer::initTree()
{
    // Set up the tree gadget.
    _tree->removeAllItems(IlvFalse);
    _tree->removeCallback(IlvTreeGadget::ExpandCallbackType(),
                          TreeItemExpanded);
    _tree->addCallback(IlvTreeGadget::ExpandCallbackType(),
                       TreeItemExpanded,
                       this);
    _tree->removeCallback(IlvTreeGadget::SelectCallbackType(),
                          TreeItemSelected);
    _tree->addCallback(IlvTreeGadget::SelectCallbackType(),
                       TreeItemSelected,
                       this);
}
```

This member function removes all the items in the tree and sets two callbacks—a selection callback that updates the sheet whenever a new folder is selected in the tree, and a callback that is triggered each time a tree item is expanded. This second callback prevents the `FileViewer` object from loading the entire file hierarchy at once, which would slow down the operation. Folders are loaded only when expanded.

> ***Note:*** *Each callback is removed prior to being added to avoid registering the same callback twice.*

For more information, see “Callbacks” in Graphic Objects.

Initializing the Sheet

The sheet is initialized with the `initSheet` member function:

```
void
FileViewer::initSheet()
{
    // Set up the sheet.
    _sheet->scrollBarShowAsNeeded(IlvTrue, IlvFalse);
    _sheet->hideScrollBar(IlvHorizontal, IlvFalse);
    _sheet->adjustLast(IlvTrue);
    _sheet->reinitialize(3, 1);
    _sheet->setNbFixedColumn(0);
    _sheet->scrollToColumn(0);
    _sheet->setExclusive(IlvTrue);
    _sheet->allowEdit(IlvFalse);

    // Create the header.
    _sheet->set(0, 0, new IlvLabelMatrixItem("Name"));
    _sheet->setItemAlignment(0, 0, IlvLeft);
    _sheet->setItemRelief(0, 0, IlvTrue);
    _sheet->setItemSensitive(0, 0, IlvFalse);
    _sheet->setItemGrayed(0, 0, IlvFalse);
    _sheet->set(1, 0, new IlvLabelMatrixItem("Size"));
    _sheet->setItemRelief(1, 0, IlvTrue);
    _sheet->setItemSensitive(1, 0, IlvFalse);
    _sheet->setItemGrayed(1, 0, IlvFalse);
    _sheet->set(2, 0, new IlvLabelMatrixItem("Type"));
    _sheet->setItemAlignment(2, 0, IlvLeft);
    _sheet->setItemRelief(2, 0, IlvTrue);
    _sheet->setItemSensitive(2, 0, IlvFalse);
    _sheet->setItemGrayed(2, 0, IlvFalse);
}
```

This member function modifies the sheet so that it has three columns and only one row in which the sheet header will be displayed. Column 1 will contain file names, column 2 will contain the size of the files, and column 3 will provide information about the file type, as shown in the following figure:

__Note:__ The items making up the header are set to insensitive, meaning that the user will not be able to select them.

Scanning Directories

The `fillTree` member function fills the tree with data once it is initialized. It reads the contents of a given directory and creates the corresponding items in the tree:

```
void
FileViewer::fillTree(IlvPathName& path, IlvTreeGadgetItem* parent)
{
    // Read the 'path' directory and insert each sub-folder as a child of
    // the item 'parent'.
    if (path.openDir()) {
        IlvPathName file;
        while (path.readDir(file)) {
            if (file.isDirectory() &&
                file.getDirectory(IlvFalse) != IlvString(".") &&
                file.getDirectory(IlvFalse) != IlvString("..")) {
                IlvTreeGadgetItem* item =
                     (IlvTreeGadgetItem*)createFileItem(file, _tree);
                item->setUnknownChildCount(IlvTrue);
                // Insert the directory 'file' into parent.
                parent->insertChild(item);
                // Compute the absolute path name.
                IlvPathName* absPathName=new IlvPathName(path);
                absPathName->merge(file);
                item->setclientData(absPathName)
             }
         }
         path.closeDir();
         // Sort the added items.
         _tree->sort(parent, 1);
      }
}
```

The `FileViewer::createFileItem` member function creates each of the tree items. These are then added to their parent item, that is, to the folder being read. The added items are then sorted by the `IlvTreeGadgetItemHolder::sort` member function.

The content of `createFileItem` is described in *Creating the Gadget Items* on page 18.

Notes:

1. *The member function* `IlvTreeGadgetItem::setUnknownChildCount` *is called for each added item to ensure that the Expand callback is invoked. For more information, see "Changing the Characteristics of an Item" in Using Common Gadgets.*
2. *The client data of an item is used to store the path object that references the directory that the item points to. This object is a cache that avoids recomputing the absolute path of the item each time.*

The fillTree method fills only one node of the tree. The whole tree is actually built by the Expand callback:

```
static void
TreeItemExpanded(IlvGraphic* g, IlvAny arg)
{
    IlvTreeGadgetItem* item = ((IlvTreeGadget*)g)->getCallbackItem();
    if (!item->hasChildren()) {
        FileViewer* viewer = (FileViewer*)arg;
        IlvPathName* path = (IlvPathName*)item->getClientData();
        viewer->updateTree(*path, item);
     }
}
```

This callback is invoked each time the user tries to expand a collapsed folder. The arg parameter is a pointer to the FileViewer object, as specified when the callback was set. The path that was stored as the item client data is retrieved and used to update the tree. Since you want to update the tree only if the folder has never been open, ensure that the item being expanded has no children before calling the updateTree method:

```
void
FileViewer::updateTree(IlvPathName& path, IlvTreeGadgetItem* item)
{
     _tree->initReDrawItems();
     // Reset the tree, if needed.
     if (!item)
         initTree();
     // Fill it using the 'path' directory.
     fillTree(path, item? item : _tree->getRoot());
     // Finally, redraw.
     _tree->reDrawItems();
}
```

This member function calls the fillTree member function and performs additional operations such as cleaning the tree and handling smart redrawing. See “Redrawing Gadget Items” in Gadget Items.

Updating the Sheet

The list of files represented by an `IlvSheet` object on the right-hand side of the window must be updated each time the item selected in the tree changes. Updating is performed by the selection callback of the `IlvTreeGadget` class.

```
static void
TreeItemSelected(IlvGraphic* g, IlvAny arg)
{
    // Retrieve the item that has triggered the callback.
    IlvTreeGadgetItem* item = ((IlvTreeGadget*)g)->getCallbackItem();
    // In the case of a selection (this callback is also called when an item
    // is deselected) update the sheet.
    if (item->isSelected()) {
        // Retrieve the path of the item.
        IlvPathName* pathname = (IlvPathName*)item->getClientData();
        // Retrieve the file viewer instance.
        FileViewer* viewer = (FileViewer*)arg;
        // Update the sheet.
        viewer->updateSheet(*pathname);
    }
}
```

This callback is invoked each time a new item is selected in the tree. Once the item that triggered the callback is retrieved by the member function `IlvTreeGadget::getCallbackItem`, it is necessary to check whether the item is

selected (since the tree selection callback is also invoked when an item is deselected). Then, the path stored in the item client data is passed to the `updateSheet` member function:

```
void
FileViewer::updateSheet(IlvPathName& path)
{
    _sheet->initReDrawItems();
    // Reset the sheet.
    initSheet();
    // Read all the files contained in 'path'.
    if (path.openDir()) {
        IlvPathName file;
        while (path.readDir(file)) {
            if (!file.isDirectory() ||
                (file.getDirectory(IlvFalse) != IlvString(".") &&
                 file.getDirectory(IlvFalse) != IlvString(".."))) {
                if (!file.isDirectory())
                    file.setDirectory(path);
                // Add the file to the sheet.
                addFile(file);
            }
        }
        path.closeDir();
    }
    // Recompute the column sizes.
    _sheet->fitWidthToSize();
    // Invalidate the whole sheet.
    _sheet->getHolder()->invalidateRegion(_sheet);
    // Finally, redraw.
    _sheet->reDrawItems();
}
```

This member function adds all the files specified in the directory path to the sheet using the `addFile` member function. It also manages the sheet redrawing:

```
void
FileViewer::addFile(const IlvPathName& file)
{
    // Create the gadget item that will be inserted into the sheet.
    IlvGadgetItem* item = createFileItem(file, _sheet);
    // Encapsulate it with a matrix item.
    IlvAbstractMatrixItem* mitem = new IlvGadgetItemMatrixItem(item);
    // Add a new row in the matrix .
    _sheet->insertRow((IlvUShort) -1);
    IlvUShort row = (IlvUShort)(_sheet->rows() - 1);

    // Set the item in the first column.
    _sheet->set(0, row, mitem);
    _sheet->resizeRow((IlvUShort)(row + 1), item->getHeight() + 1);
    _sheet->setItemAlignment(0, row, _sheet->getItemAlignment(0, 0));
    // File Size in the second column.
    _sheet->setItemSensitive(1, row, IlvFalse);
    _sheet->setItemGrayed(1, row, IlvFalse);
    _sheet->setItemAlignment(1, row, _sheet->getItemAlignment(1, 0));
    ifstream ifile(file.getString(), IlvBinaryInputStreamMode);
    if (!(!ifile)) {
        ifile.seekg(0, ios::end);
        streampos length = ifile.tellg();
        mitem = new IlvIntMatrixItem(length);
        _sheet->set(1, row, mitem);
    }
    // File Type in the third column.
    mitem = new IlvLabelMatrixItem(file.isDirectory()
                                   ? "File Folder"
                                   : "File");
    _sheet->setItemSensitive(2, row, IlvFalse);
    _sheet->setItemGrayed(2, row, IlvFalse);
    _sheet->setItemAlignment(2, row, _sheet->getItemAlignment(2, 0));
    _sheet->set(2, row, mitem);
}
```

This member function creates the item corresponding to the file parameter provided as its argument and adds it to the first column of the sheet. Then, it retrieves the file size and puts it in the second column as an `IlvIntMatrixItem` object. Finally, it puts the file type in the third column as an `IlvLabelMatrixItem` object.

Both the second and third column items are set to insensitive, meaning that the user will not be able to select them.

> ***Note:*** *The file names in the first column of the sheet are created with the same method as the one used for building the tree.*

Creating the Gadget Items

The member function `FileViewer::createFileItem` creates the items:

```
IlvGadgetItem*
FileViewer::createFileItem(const IlvPathName& file,
                           const IlvGadgetItemHolder* holder) const
{
    // Compute the item label.
    IlvString filename = file.isDirectory()
        ? file.getString()
        : file.getBaseName();
    // If file is a directory, remove the trailing '/'.
    if (file.isDirectory())
        filename.remove(filename.getLength() - 1);
    return
        holder->createItem(filename,
                           0,
                           getBitmap(file, IlvGadgetItem::BitmapSymbol()),
                           getBitmap(file,
                                     IlvGadgetItem::SelectedBitmapSymbol()));
}
```

A gadget item can include both a picture and a label. For a detailed description of gadget items, see Gadget Items. The item label is computed from the file parameter. Items are assigned two bitmaps—one that represents them when they are selected and one that represents them when they are not selected.

Then, the item is created using the factory member function `IlvGadgetItemHolder::createItem`, which is called on the holder instance provided as its parameter. See “Creating Gadget Items” in Gadget Items.

In the section *Scanning Directories* on page 13, the items of both the tree and the sheet are created by calling this method. This is because the classes `IlvTreeGadget` and `IlvSheet` both inherit from the `IlvGadgetItemHolder` class. Calling the `createFileItem` member function with an instance of the `IlvTreeGadget` class as its holder parameter returns an instance of `IlvTreeGadgetItem`, whereas calling it with an instance of `IlvSheet` returns an instance of the `IlvGadgetItem` class.

The member function `IlvGadgetItemHolder::createItem` takes two bitmaps as its third and fourth parameters. This makes it easy to create an item with predefined bitmaps.

However, the call to `createItem` could have been replaced by the following code, which shows how to assign a bitmap to a gadget item:

```
IlvGadgetItem* item = holder->createItem(filename);
item->setBitmap(IlvGadgetItem::BitmapSymbol(),
                getBitmap(file, IlvGadgetItem::BitmapSymbol()));
item->setBitmap(IlvGadgetItem::SelectedBitmapSymbol(),
                getBitmap(file, IlvGadgetItem::SelectedBitmapSymbol()));
return item;
```

The getBitmap method used here returns the bitmap corresponding to the pair file type/item state. Its code is very simple, as can be seen here:

```
IlvBitmap*
FileViewer::getBitmap(const IlvPathName& file,
                      const IlvSymbol* state) const
{
     if (state == IlvGadgetItem::BitmapSymbol())
         return getBitmap(file.isDirectory()
                          ? folderBm
                          : fileBm);
     else
         return getBitmap(file.isDirectory()
                          ? sfolderBm
                          : sfileBm);
}
```

folderBm, fileBm, sfolderBm, and sfileBm are defined symbols, where:

- folderBm is the name of the bitmap used for folders.
- fileBm is the name of the bitmap used for nonselected files.
- sfolderBm is the name of the bitmap used for selected folders.
- sfileBm is the name of the bitmap used for selected files.

Instantiating the File Viewer

The FileViewer class is now complete. To instantiate it in the application, you must add a call to the member function FileViewerApplication::configureApplication in FileViewerApplication::makePanels, which was defined in the section *Creating the Main Window* on page 8.

```
void
FileViewerApplication::makePanels()
{
    // Initialize the main window.
    initMainWindow();
    // Initialize the application.
    configureApplication();
    // Show it.
    getMainWindow()->show();
}
```

The FileViewerApplication::configureApplication member function instantiates and initializes the FileViewer class:

```
void
FileViewerApplication::configureApplication()
{
    FileViewer* viewer = createFileViewer(getMainWindow());
    FileViewerApplication::SetFileViewer(getMainWindow(), viewer);
    viewer->init(IlvPathName("/"));
}
```

> ***Note:*** *Calling* `FileViewerApplication::SetFileViewer` *connects the main window to the viewer. You can retrieve the file viewer connected to a specific window with* `FileViewerApplication::GetFileViewer`.

The factory method `createFileViewer` returns a new instance of the `FileViewer` class:

```
FileViewer*
FileViewerApplication::createFileViewer(FileViewerWindow* window) const
{
    return new FileViewer(window->getDirectoryHierarchy(),
                          window->getFileList());
}
```

Previewing the Application

This is the end of Step 2. Your application window should look like this:

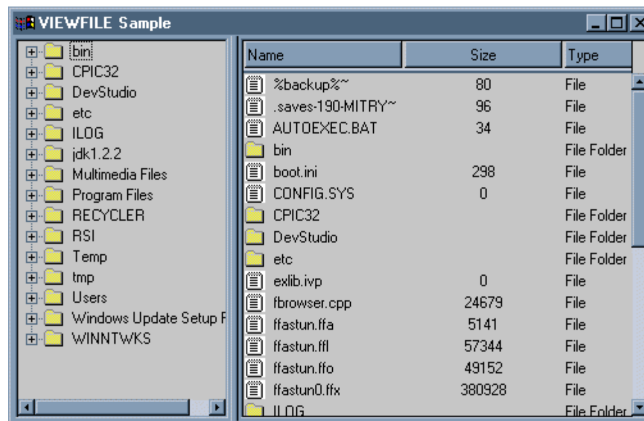


Figure 1.4 *The ViewFile Application After Step2*

Step 3: Adding Docking Bars

This step explains how to add docking bars (menu bars and toolbars) to the ViewFile application. For information on docking panes, see Docking Panes and Containers.

This step shows you how to perform the following tasks:

- *Preparing the Main Window for Docking*
- *Creating the Docking Toolbar*

- *Creating the Docking Menu Bar*
- *Integrating Changes into the Application*
- *Previewing the Application*

All the changes made in this step are reflected in the file `viewfile.cpp`.

Preparing the Main Window for Docking

Docking panes, and hence docking toolbars and menu bars, must be docked to dockable containers. Therefore, the application main window must be modified to allow docking operations.

Modifying the Main Window

`IlvDockableContainer`, a subclass of `IlvPanedContainer`, is the base class for handling docking panes. In this step, the main window is defined as an `IlvDockableMainWindow` object in the member function `FileViewerApplication::createMainWindow` (see *Creating the Main Window* on page 8). `IlvDockableMainWindow` is a subclass of `IlvDockableContainer` that implements a standard layout for applications incorporating docking panes. See the section “Building a Standard Application with Docking Panes” in Docking Panes and Containers.

```
IlvContainer*
FileViewerApplication::createMainWindow(const IlvRect& rect) const
{
    return new IlvDockableMainWindow(getDisplay(),
                                      getName(),
                                      getName(),
                                      rect,
                                      0,
                                      IlvFalse);
}
```

It is now possible to add docking panes to the application.

Preparing the Panes Creation

The member function `FileViewerApplication::makePanels` must be modified to include calls to `initPanes`, which creates the panes, and `updatePanes`, which updates the

paned container layout. See the section “Modifying the Layout of a Paned Containers” in Panes.

```
void
FileViewerApplication::makePanels(){
    // Initialize the main window.
    initMainWindow();
    // Initialize the panes.
    initPanes();
    // Initialize the application.
    configureApplication();
    // Update the main window layout.
    getMainWindow()->updatePanes(IlvTrue);
    // Show it.
    getMainWindow()->show();
}
```

The `initPanes` member function calls `initMenuBar`, which creates the menu bar and `initToolBar`, which creates the toolbar.

```
void
FileViewerApplication::initPanes()
{
    // Initialize the menu bar.
    initMenuBar();
    // Initialize the toolbar.
    initToolBar();
}
```

Creating the Docking Toolbar

The `initToolBar` member function creates the toolbar:

```
void
FileViewerApplication::initToolBar()
{
    IlvToolBar* toolbar = new IlvToolBar(getDisplay(), IlvPoint(0, 0));
    // Item Up One Level.
    toolbar->insertBitmap(getBitmap("upBm"));
    toolbar->getItem(0)->setCallback(UpOneLevel);
    toolbar->getItem(0)->setClientData(this);
    toolbar->getItem(0)->setToolTip("Up One Level");
    // Encapsulate the toolbar into a graphic pane.
    IlvGraphicPane* toolbarPane = new IlvAbstractBarPane("Toolbar", toolbar);
    // Add the pane to the application on top of the main workspace.
    getMainWindow()->addRelativeDockingPane(toolbarPane,
                                            IlvDockableMainWindow::
                                            GetMainWorkspaceName(),
                                            IlvTop);
}
```

The toolbar contains only one item, the “Up One Level” item. When selected, this item tries to explore the parent of the selected folder. (The items of the selected folder are displayed in the sheet on the right side of the file browser.) The “Up One Level” item is associated with a

tooltip. The toolbar is encapsulated in a special graphic pane of type `IlvAbstractBarPane`, which is added to the main window, just above the workspace.

For more information, see the sections “Using Tooltips in a Toolbar” in Menus, Menu Bars, and Toolbars and “Using the IlvAbstractBarPane Class” in Docking Panes and Containers.

***Note:** The client data of the item is set to `this`, making it possible to retrieve a pointer to the application from its callback function. (Remember that the client data is the second parameter of the callback invoked when the item is activated.)*

```
static void
UpOneLevel(IlvGraphic* g, IlvAny arg)
{
    // Retrieve a pointer on the application.
    FileViewerApplication* application = (FileViewerApplication*)arg;
    // Retrieve a pointer on the main window.
    FileViewerWindow* window = (FileViewerWindow*)
        application->getMainWindow()->getMainWorkspaceViewPane()->getView();
    // Retrieve a pointer on the file viewer.
    FileViewer* viewer = FileViewerApplication::GetFileViewer(window);
    if (viewer) {
        // Find the last selected item of the tree.
        IlvTreeGadgetItem* item =
            viewer->getTreeGadget()->getLastSelectedItem();
        // And select its parent.
        if (item && item->getParent() && item->getParent()->getParent())
            viewer->getTreeGadget()->selectItem(item->getParent(),
                                                IlvTrue,
                                                IlvTrue,
                                                IlvTrue);
        }
}
```

Creating the Docking Menu Bar

The `initMenuBar` member function creates the menu bar:

```
void
FileViewerApplication::initMenuBar()
{
    // The menu bar is in fact an IlvToolBar.
    IlvToolBar* menubar = new IlvToolBar(getDisplay(), IlvPoint(0, 0));
    // Add two items
    menubar->addLabel("File");
    menubar->addLabel("Help");
    // Create the pane that will encapsulate the menu bar.
    IlvGraphicPane* menubarPane = new ApplicationMenuBarPane("Menu Bar",
                                                             menubar);
    // Change the mode of the menu bar to make it show items on several
    // rows, if needed.
    menubar->setConstraintMode(IlvTrue);
    // Add the pane to the application on top of the main workspace.
    getMainWindow()->addRelativeDockingPane(menubarPane,
                                            IlvDockableMainWindow::
                                            GetMainWorkspaceName(),
                                            IlvTop);
    // Now fill the menus with popup menus.
    IlvPopupMenu* menu;
    // Menu File: New / Separator / Exit.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("Exit");
    menu->getItem(0)->setCallback(ExitApplication);
    menu->getItem(0)->setClientData(this);
    menubar->getItem(0)->setMenu(menu, IlvFalse);
    // Menu Help: About.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("About");
    menubar->getItem(1)->setMenu(menu, IlvFalse);
    menu->getItem(0)->setCallback(ShowAboutPanel);
    menu->getItem(0)->setClientData(this);
}
```

The menu bar is created and initialized with two items: File and Help. It is then encapsulated into a pane of the `ApplicationMenuBarPane` class, a subclass of `IlvAbstractBarPane` class that changes the label orientation according to the bar orientation.

See the section “Customizing Docking Bars” in Docking Panes and Containers.

The Help item of the menu bar has a submenu that contains the About item. This item is attached to a callback that displays a dialog box showing the application name:

```
void
FileViewerApplication::showVersion()
{
    IlvIInformationDialog dialog(getDisplay(), "VIEWFILE Tutorial");
    dialog.moveToMouse();
    dialog.showModal();
}
```

For more information, see Dialogs.

Integrating Changes into the Application

The last step consists of modifying the member function `FileViewerApplication::configureApplication` to include the view of the file viewer in the main workspace of the application main window. This workspace is represented by a view pane that is created by default and can be retrieved using the `IlvDockableMainWindow::getMainWorkspaceViewPane`.

See "Using the IlvDockableMainWindow Class" in Docking Panes and Containers.

```
void
FileViewerApplication::configureApplication()
{
    // Create the file viewer window.
    FileViewerWindow* window = createFileViewerWindow(getMainWindow(),
                                                  IlvRect(0, 0, 400, 200));
    // Replace the view of the main workspace pane with the file viewer window.
    getMainWindow()->getMainWorkspaceViewPane()->setView(window);
    // Create the file viewer using the file viewer window.
    FileViewer* viewer = createFileViewer(window);
    // Link the file viewer with its window.
    // This is used in the UpOneLevel callback.
    SetFileViewer(window, viewer);
    // Initialize the file viewer.
    viewer->init(IlvPathName("/"));
}
```

The factory member function `createFileViewerWindow` creates a `FileViewerWindow` object as a subview of the provided view. The created view is set as the view pane representing the main workspace. Finally, the `FileViewer` object is created and initialized, using the same method as in the previous step.

Previewing the Application

This is the end of Step 3. Your application window should look like this:

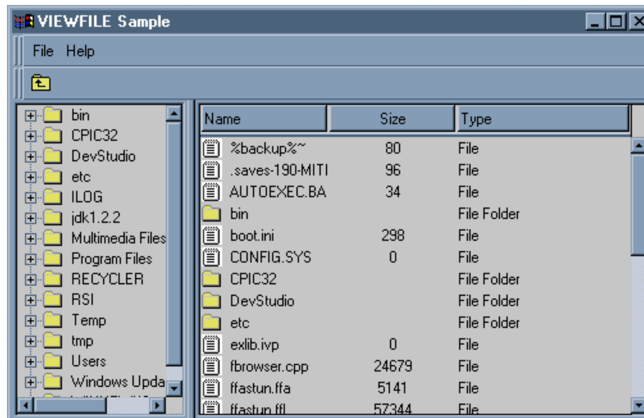


Figure 1.5 *The ViewFile Application After Step 3*

Step 4: Adding View Frames

This last step explains how to add view frame support to the ViewFile application. Including view frames into the application allows you to display and browse through several hierarchical lists of files at the same time. For more information, see View Frames.

All changes made in this step are reflected in the file `viewfile.cpp`.

This step shows you how to perform the following tasks:

- *Choosing the Desktop View*
- *Creating View Frames*
- *Adding New Menus and Items to View Frames*
- *Previewing the Application*

Choosing the Desktop View

View frames are displayed inside a desktop view. The first operation consists of choosing the desktop view where view frames will be displayed. The best place to put the desktop view is the main workspace view pane, where the unique file viewer was displayed in *Step 3: Adding Docking Bars* on page 20. The desktop manager that will manage all the view frames can be created from this view.

Creating the Desktop Manager

The `FileViewerApplication::makePanels` member function must be modified to include a call to `initDesktopManager`, which creates the desktop manager using the main workspace view pane.

```
void
FileViewerApplication::makePanels()
{
    // Initialize the main window.
    initMainWindow();

    // Initialize the desktop manager.
    initDesktopManager();

    // Initialize the panes.
    initPanes();

    // Initialize the application.
    configureApplication();

    // Update the main window layout.
    getMainWindow()->updatePanes(IlvTrue);

    // Show it.
    getMainWindow()->show();
}
```

Here is the detail of `initDesktopManager`:

```
void
FileViewerApplication::initDesktopManager()
{
    createDesktopManager(getMainWindow()->
    getMainWorkspaceViewPane()->getView());
}
```

The factory member function `FileViewerApplication::createDesktopManager` returns an instance of the `IlvDesktopManager` class:

```
IlvDesktopManager*
FileViewerApplication::createDesktopManager(IlvView* view) const
{
    return new IlvDesktopManager(view);
}
```

Initializing the Desktop Manager

You must also modify `FileViewerApplication::configureApplication` to initialize the desktop manager:

```
void
FileViewerApplication::configureApplication()
{
    // The desktop manager is maximized by default.
    getDesktopManager()->
       makeMaximizedStateButtons((IlvToolBar*)
                                 ((IlvGraphicPane*)getMainWindow()->
                                   getPane("Menu Bar", IlvTrue))->getObject());
    getDesktopManager()->maximize(0);

    // Create a frame initialized at "/".
    getDesktopManager()->setCurrentFrame(createNewFrame(IlvRect(0,
                                                   0,
                                                   400,
                                                   200), "/"));
}
```

`IlvDesktopManager::makeMaximizedStateButtons` specifies where the buttons of the view frame should appear when it is maximized.

See "Maximized View Frames" in View Frames.

The buttons are displayed inside the menu bar, as shown here:

Then, the desktop manager is maximized, and a default frame is created by `FileViewerApplication::createNewFrame`.

Creating View Frames

The member function `FileViewerApplication::createNewFrame` creates a new frame that encapsulates the file viewer and its associated window:

```
IlvViewFrame*
FileViewerApplication::createNewFrame(const IlvRect& rect,
const char* path) const
{
    // Create a view frame in the desktop manager view.
    IlvViewFrame* vframe = new IlvViewFrame(getDesktopManager()->getView(),
                                            path,
                                            rect,
                                            IlvFalse);
     vframe->setDestroyCallback(DestroyFrame);
     // Create a file viewer window inside the view frame.
     FileViewerWindow* viewerWindow = createFileViewerWindow(vframe, rect);
     // Create the file viewer in the file viewer window.
     FileViewer* viewer = createFileViewer(viewerWindow);
     // Associate the viewer window with the viewer.
     SetFileViewer(viewerWindow, viewer);
     // Initialize the file viewer.
     viewer->init(IlvPathName(path));
     return vframe;
}
```

The member function `IlvView::setDestroyCallback` sets a destroy callback that handles the deletion of the created frame. See “Closing View Frames” in View Frames.

Then, the file viewer window is created as a subview of the view frame. See “Creating a Client View” in View Frames. Finally, a file viewer is connected to the view frame and initialized.

Adding New Menus and Items to View Frames

To add new menu items to the menu bar, you must modify the member function

FileViewerApplication::initMenuBar as follows:

```
void
FileViewerApplication::initMenuBar()
{
    // The menu bar is in fact an IlvToolBar.
    IlvToolBar* menubar = new IlvToolBar(getDisplay(), IlvPoint(0, 0));
    // Add three items.
    menubar->addLabel("File");
    menubar->addLabel("Windows");
    menubar->addLabel("Help");
    // Create the pane that will encapsulate the menu bar.
    IlvGraphicPane* menubarPane = new ApplicationMenuBarPane("Menu Bar",
                                                             menubar);
    // Change the mode of the menu bar to make it show items on several
    // rows, if needed.
    menubar->setConstraintMode(IlvTrue);
    // Add the pane to the application on top of the main workspace.
    getMainWindow()->addRelativeDockingPane(menubarPane,
                                            IlvDockableMainWindow::
                                            GetMainWorkspaceName(),
                                            IlvTop);
    // Now fill the menus with popup menus.
    IlvPopupMenu* menu;
    // Menu File: New / Separator / Exit.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("New (Ctrl+N)");
    menu->getItem(0)->setBitmap(getBitmap("newBm"));
    menu->getItem(0)->setCallback(AddNewFrame);
    menu->getItem(0)->setClientData(this);
    menu->getItem(0)->setAcceleratorText("Ctrl+N");
    menu->getItem(0)->setAcceleratorKey(IlvCtrlChar('N'));
    menu->addItem(IlvMenuItem());
    menu->addLabel("Exit");
    menu->getItem(2)->setCallback(ExitApplication);
    menu->getItem(2)->setClientData(this);
    menubar->getItem(0)->setMenu(menu, IlvFalse);
    // Menu Windows: Cascade / Tile Horizontally / Tile Vertically.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("Cascade");
    menu->getItem(0)->setCallback(CascadeFrames);
    menu->getItem(0)->setClientData(this);
    menu->addLabel("Tile Horizontally");
    menu->getItem(1)->setCallback(TileHorizontallyFrames);
    menu->getItem(1)->setClientData(this);
    menu->addLabel("Tile Vertically");
    menu->getItem(2)->setCallback(TileVerticallyFrames);
    menu->getItem(2)->setClientData(this);
    menubar->getItem(1)->setMenu(menu, IlvFalse);
    // Menu Help: About.
    menu = new IlvPopupMenu(getDisplay());
    menu->addLabel("About");
    menubar->getItem(2)->setMenu(menu, IlvFalse);
    menu->getItem(0)->setCallback(ShowAboutPanel);
    menu->getItem(0)->setClientData(this);
}
```

The menu bar has a Windows menu that contains the sub-items shown in the following figure:

The callbacks used by the Windows menu invoke the corresponding member functions on the desktop manager:

```
static void
CascadeFrames(IlvGraphic* g, IlvAny arg)
{
    FileViewerApplication* application = (FileViewerApplication*)arg;
    application->getDesktopManager()->cascadeFrames();
}

static void
TileHorizontallyFrames(IlvGraphic* g, IlvAny arg)
{
    FileViewerApplication* application = (FileViewerApplication*)arg;
    application->getDesktopManager()->tileFrames(IlvHorizontal);
}

static void
TileVerticallyFrames(IlvGraphic* g, IlvAny arg)
{
    FileViewerApplication* application = (FileViewerApplication*)arg;
    application->getDesktopManager()->tileFrames(IlvVertical);
}
```

The member function `FileViewerApplication::initToolBar` has also been modified to include a New item that creates a new frame when selected:

```
void
FileViewerApplication::initToolBar()
{
    IlvToolBar* toolbar = new IlvToolBar(getDisplay(), IlvPoint(0, 0));
    // Item New.
    toolbar->insertBitmap(getBitmap("newBm"));
    toolbar->getItem(0)->setCallback(AddNewFrame);
    toolbar->getItem(0)->setClientData(this);
    toolbar->getItem(0)->setToolTip("New");
    // Separator.
    toolbar->addItem(IlvMenuItem());
    // Item Up One Level.
    toolbar->insertBitmap(getBitmap("upBm"));
    toolbar->getItem(2)->setCallback(UpOneLevel);
    toolbar->getItem(2)->setClientData(this);
    toolbar->getItem(2)->setToolTip("Up One Level");
    // Encapsulate the toolbar into a graphic pane.
    IlvGraphicPane* toolbarPane = new IlvAbstractBarPane("Toolbar", toolbar);
    // Add the pane to the application on top of the main workspace.
    getMainWindow()->addRelativeDockingPane(toolbarPane,
                                        IlvDockableMainWindow::
                                        GetMainWorkspaceName(),
                                        IlvTop);
}
```

The callback triggered by the new item invokes the member function `FileViewerApplication::createNewFrame` and activates the new frame:

```
static void
AddNewFrame(IlvGraphic* g, IlvAny arg)
{
    FileViewerApplication* application = (FileViewerApplication*)arg;
    IlvViewFrame* vframe =
       application->createNewFrame(IlvRect(0, 0, 400, 200), "/");
    application->getDesktopManager()->setCurrentFrame(vframe);
}
```

> ***Note:*** *The New item is associated with an accelerator key allowing the user to create a new frame by pressing the Ctrl+N key combination. See "Associating Accelerators on Menu Items" in Menus, Menu Bars, and Toolbars.*

Previewing the Application

You have completed this tutorial. The application window should look like this:

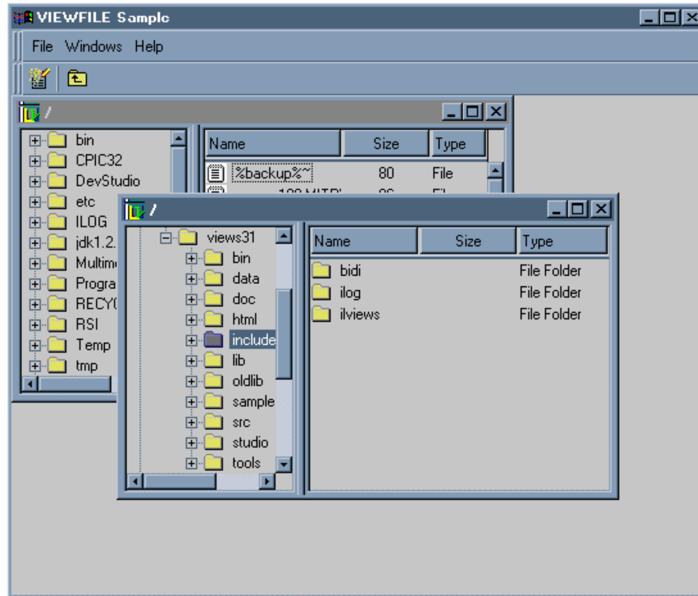


Figure 1.6 *The Final ViewFile Application*

C H A P T E R

2

Customizing Gadgets

This tutorial shows you how to create new gadgets. Creating a new gadget class is very similar to creating a new graphic class (that is, a subclass of `IlvGraphic`), but it requires more work. For example, a gadget must define its own behavior (through the `IlvGadget::handleEvent` method) and must define the way it will deal with keyboard focus.

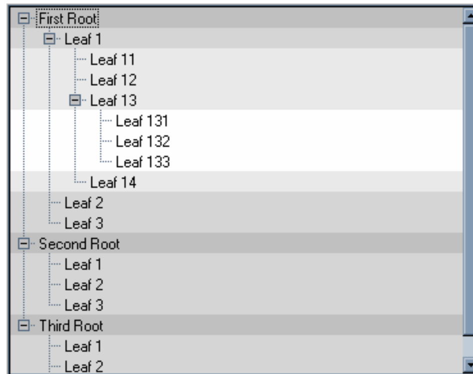
This tutorial has three steps:

- *Step 1: Extending a Gadget by Changing its Graphical Appearance*
- *Step 2: Extending a Gadget by Changing its Behavior*
- *Step 3: Creating a Composite Gadget*

Step 1: Extending a Gadget by Changing its Graphical Appearance

This first step describes how to create a subclass of an existing gadget to change its graphical appearance.

In this step the `IlvTreeGadget` class will be extended. Its drawing will be modified to allow the user to change the color of the children of an item, as shown in the following figure:



This step shows you how to perform the following tasks:

- *Creating a Subclass of an Existing Gadget*
- *Adding an API to the New Gadget Class*
- *Modifying How the New Gadget is Drawn*
- *Testing the New Gadget Class*

Creating a Subclass of an Existing Gadget

Creating a new subclass of an existing gadget is similar to creating a subclass of any existing `IlvGraphic` class. The main steps to properly create a subclass of an `IlvGraphic` object are described here.

Declare the Subclass

Declare your subclass as a subclass of the chosen graphic. In this tutorial, `IlvTreeGadget` is subclassed to create a new object that can display colored items. This subclass will be called `ColoredTreeGadget`. Here is the declaration of the `ColoredTreeGadget` class:

```
class ColoredTreeGadget
: public IlvTreeGadget {
public:
    ColoredTreeGadget(IlvDisplay*        display,
                      const IlvRect&     rect,
                      IlvUShort          thickness = IlvDefaultGadgetThickness,
                      IlvPalette*        palette   = 0);
    virtual ~ColoredTreeGadget();
};
```

This declaration is made up of a constructor and the destructor of the `ColoredTreeGadget` class.

Add the registration of the `ColoredTreeGadget` to allow dynamic typing as well as persistence:

```
DeclareTypeInfo();
DeclareIOConstructors(ColoredTreeGadget);
DeclareGraphicAccessors();
```

The `DeclareGraphicAccessors` macro declares the `queryValue` and `applyValue` methods to allow the new object to be scriptable.

Implement the Methods

The implementation of the `ColoredTreeGadget` methods that have been declared is shown here.

First, the constructor is implemented as follows:

```
ColoredTreeGadget::ColoredTreeGadget(IlvDisplay*        display,
                                     const IlvRect&     rect,
                                     IlvUShort          thickness,
                                     IlvPalette*        palette)
: IlvTreeGadget(display, rect, thickness, palette)
{
}
```

The copy constructor, declared by the `DeclareIOContructors` macro:

```
ColoredTreeGadget::ColoredTreeGadget(const ColoredTreeGadget& source)
: IlvTreeGadget(source)
{
}
```

Then, the IO constructor, also declared by the `DeclareIOConstructors` macro:

```
ColoredTreeGadget::ColoredTreeGadget(IlvInputFile& is, IlvPalette* palette)
: IlvTreeGadget(is, palette)
{
}
```

The `write` method, declared by the `DeclareTypeInfo` macro:

```
void
ColoredTreeGadget::write(IlvOutputFile& os) const
{
    IlvTreeGadget::write(os);
}
```

The destructor:

```
ColoredTreeGadget::~ColoredTreeGadget()
{
}
```

The accessor related methods, declared by the `DeclareGraphicAccessors` macro:

```
IlvValue&
ColoredTreeGadget::queryValue(IlvValue& value) const
{
    return IlvTreeGadget::queryValue(value);
}

IlvBoolean
ColoredTreeGadget::applyValue(const IlvValue& value)
{
    return IlvTreeGadget::applyValue(value);
}

void
ColoredTreeGadget::GetAccessors(const IlvSymbol* const** a,
                                const IlvValueTypeClass* const** t,
                                IlvUInt& c)
{
}
```

Then, the implementation of the `copy` and `read` methods are given by the `IlvPredefinedIOMembers` macro:

```
IlvPredefinedIOMembers(ColoredTreeGadget)
```

Finally, the class is registered as a subclass of the `IlvTreeGadget` class:

```
IlvRegisterClass(ColoredTreeGadget, IlvTreeGadget);
```

These items are necessary to properly register the `ColoredTreeGadget` class. Although many of these methods are empty, they will be filled in the next tasks.

Adding an API to the New Gadget Class

The declaration of the `ColoredTreeGadget` class can be found in the `coltree.h` file, and its implementation can be found in the `coltree.cpp` file.

The `ColoredTreeGadget` class requires the following:

- A way to associate a tree item with a specific color. This color will be used to draw the background of the children of this item.
- A way to enable or disable the coloring of the tree items.
- Adding accessors to it.

Associating an Item With a Color

In order to associate a tree item with a color, the `setChildrenBackground` method has been added. It uses a property set on the tree item to store the color:

```
void
ColoredTreeGadget::setChildrenBackground(IlvTreeGadgetItem* item,
                                         IlvColor* color,
                                         IlvBoolean redraw)
{
    // Retrieve the old color.
    IlvPalette* oldPalette =
             (IlvPalette*)item->getProperty(GetChildrenBackgroundSymbol());
    // Compute the new color.
    IlvPalette* palette = color
                        ? getDisplay()->getPalette(0, color)
                        : 0;
    // Lock it.
    if (palette)
        palette->lock();
    // Unlock the old one.
    if (oldPalette)
        oldPalette->unLock();
    // Set the property to the item.
    item->setProperty(GetChildrenBackgroundSymbol(), (IlvAny)palette);
    // Redraw if asked.
    if (redraw)
        reDraw();
}
```

The `GetChildrenBackgroundSymbol` static method returns a symbol that identifies the property set on the item. Its definition is simple:

```
static IlvSymbol*
GetChildrenBackgroundSymbol()
{
    // This symbol is used to connect a tree gadget item to the color of its
    // children.
    static IlvSymbol* symbol = IlvGetSymbol("ChildrenBackground");
    return symbol;
}
```

The `getChildrenBackground` method returns the color associated with a given tree item.

```
IlvColor*
ColoredTreeGadget::getChildrenBackground(const IlvTreeGadgetItem* item) const
{
    // Returns the color stored in the property list of the specified item.
    IlvPalette* palette =
        (IlvPalette*)item->getProperty(GetChildrenBackgroundSymbol());
    return palette
           ? palette->getForeground()
           : 0;
}
```

Enable/Disable the Coloring

The code that will allow the user to enable or disable the coloring of a tree item must be implemented. To do this, a protected member variable is added to the `ColoredTreeGadget` class:

```
protected:
    IlvBoolean _drawChildrenBg;
```

This Boolean value will be initialized by the constructors to `IlvTrue` by default. Here are the member functions that give access to the `_drawChildrenBg` member variable:

```
IlvBoolean isDrawingChildrenBackground() const;
void drawChildrenBackground(IlvBoolean value,
                            IlvBoolean redraw = IlvTrue);
```

Since the `_drawChildrenBg` member variable must be saved, both the IO constructor and the `write` method must be modified:

```
ColoredTreeGadget::ColoredTreeGadget(IlvInputFile& is, IlvPalette* palette)
: IlvTreeGadget(is, palette),
  _drawChildrenBg(IlvTrue)
{
    // Read the _drawChildrenBg flag.
    int drawChildrenBg;
    is.getStream() >> drawChildrenBg;
    _drawChildrenBg = (IlvBoolean)drawChildrenBg;
}

void
ColoredTreeGadget::write(IlvOutputFile& os) const
{
    IlvTreeGadget::write(os);
    // Write the _drawChildrenBg flag.
    os.getStream() << IlvSpc() << (int)_drawChildrenBg << IlvSpc();
}
```

Adding Accessors

Adding accessors to objects makes them available for scripting. The following description shows an example of an accessor to the `_drawChildrenBg` member variable.

First, the symbol that will be used to access this variable is defined. This is done through the static function `GetDrawChildrenBackgroundSymbol`.

```
static IlvSymbol*
GetDrawChildrenBackgroundSymbol()
{
    // This symbol is used to access to drawChildrenBackground accessor of
    // the colored tree gadget.
    static IlvSymbol* symbol = IlvGetSymbol("drawChildrenBackground");
    return symbol;
}
```

Then, the new accessor is registered in `GetAccessors`:

```
void
ColoredTreeGadget::GetAccessors(const IlvSymbol* const** a,
                                const IlvValueTypeClass* const** t,
                                IlvUInt& c)
{
    DeclareAccessor(GetDrawChildrenBackgroundSymbol(),
                    IlvValueBooleanType,
                    a,
                    t,
                    c);
}
```

The `queryValue` method is called when the value of the accessor is queried.

```
IlvValue&
ColoredTreeGadget::queryValue(IlvValue& value) const
{
    if (value.getName() == GetDrawChildrenBackgroundSymbol())
        return value = isDrawingChildrenBackground();
    else
        return IlvTreeGadget::queryValue(value);
}
```

The `applyValue` method is called when the value of the accessor is modified.

```
IlvBoolean
ColoredTreeGadget::applyValue(const IlvValue& value)
{
    if (value.getName() == GetDrawChildrenBackgroundSymbol()) {
        drawChildrenBackground((IlvBoolean)value, IlvFalse);
        return IlvTrue;
    } else
        return IlvTreeGadget::applyValue(value);
}
```

Modifying How the New Gadget is Drawn

The next step is to change the way `ColoredTreeGadget` is drawn. The standard way to change the drawing of a graphical object is to override its `draw` method defined at the `IlvGraphic` level:

```
virtual void draw(IlvPort*               dst,
                  const IlvTransformer* t = 0,
                  const IlvRegion*      clip = 0) const = 0;
```

However, the `IlvTreeGadget` is a complex object and has several other methods that can help customize its drawing. Here, you want to draw the background before each item is

drawn. You can use the `IlvGadgetItemHolder::drawGadgetItem` method. It is called by the tree to draw each item.

```
void
ColoredTreeGadget::drawGadgetItem(const IlvGadgetItem* item,
                                  IlvPort* port,
                                  const IlvRect& rect,
                                  const IlvTransformer* t,
                                  const IlvRegion* clip) const
{
    if (isDrawingChildrenBackground()) {
         // Check if the item being drawn has a special palette.
         IlvPalette* palette = getBackgroundPalette((IlvTreeGadgetItem*)item);
         if (palette) {
             // Compute the visible bounding box.
             IlvRect bbox;
             visibleBBox(bbox, t);
             // Move and resize it to match the item bounding box.
             bbox.y(rect.y());
             bbox.h(rect.h());
             if (clip)
                 palette->setClip(clip);
             port->fillRectangle(palette, bbox);
             if (clip)
                 palette->setClip();
        }
    }
    // Draw the item.
    IlvTreeGadget::drawGadgetItem(item, port, rect, t, clip);
}
```

The `getBackgroundPalette` method is used to retrieve the palette used to draw the background of a tree item.

```
IlvPalette*
ColoredTreeGadget::getBackgroundPalette(const IlvTreeGadgetItem* item) const
{
    // Returns the palette that will be used to draw the background of 'item'
    // This information is stored in its parent
    if (item->getParent()) {
        IlvPalette* palette = (IlvPalette*)
            item->getParent()->getProperty(GetChildrenBackgroundSymbol());
        if (!palette)
            palette = getBackgroundPalette(item->getParent());
        return palette;
    } else
        return 0;
}
```

The `getBackgroundPalette` method tries to find the first parent of ‘item’ that has been set as a background palette. This means that a tree item inherits its color from its parent—changing the background color of an item may not only change the background color of its children, but also the background color of the children of its children, and so on.

Testing the New Gadget Class

The file `main.cpp` contains the code to test the `ColoredTreeGadget` class. It reads the file `coltree.ilv`, which contains the description of a `ColoredTreeGadget` instance, and changes the color of all items according to their level.

```
    // Read the file that contains the colored tree.
    container->readFile("../doc/gadgets/tutorials/custgad/data/coltree.ilv");

    // Retrieve the tree.
    ColoredTreeGadget* tree = (ColoredTreeGadget*)container->getObject("Tree");

    // Set the background of the tree to gray.
    // This color will be used as the reference color to compute the children
    // colors.
    tree->setBackground(display->getColor("gray"));

    // Now change the color of each level of items.
    tree->applyToItems(ChangeColor, (IlvAny)tree);
```

The `applyToItems` method applies the `ChangeColor` function to every item in the tree. Here is the description of this `ChangeColor` function, which calls the `ColoredTreeGadget::setChildrenBackground` method:

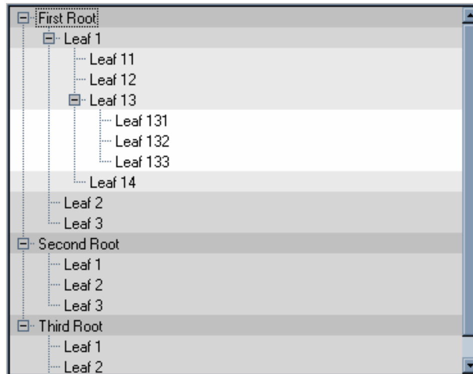
```
static IlvBoolean
ChangeColor(IlvGadgetItem* item, IlvAny arg)
{
    // The argument is a pointer to the ColoredTreeGadget instance.
    ColoredTreeGadget* tree = (ColoredTreeGadget*)arg;
    // Change the background of the children of 'item'.
    tree->setChildrenBackground((IlvTreeGadgetItem*)item,
                                GetChildrenColor((IlvTreeGadgetItem*)item,
                                                 tree->getBackground()),
                                                 IlvFalse);
    // Continue.
    return IlvTrue;
}
```

Another static function is used to compute the color using a tree item and a reference color:

```
static IlvColor*
GetChildrenColor(IlvTreeGadgetItem* item, IlvColor* color)
{
    // Get the item level to choose the right color.
    IlvUInt level = item->getLevel();
    // Compute the HSV components of the reference color.
    IlvFloat h, s, v;
    color->getHSV(h, s, v);
    // Increase the V component.
    v = (IlvFloat)IlvMin((IlvFloat)1., (IlvFloat)(v + level*.08));
    // Return the new color.
    return color->getDisplay()->getColor(h, s, v);
}
```

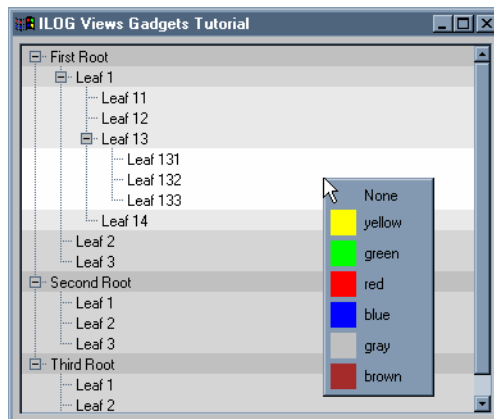
This function uses the level of the item and the specified reference color to compute a new color. This new color is computed by changing the V component of the reference color in the HSV model.

This is the end of Step 1. The test program should look like this:



Step 2: Extending a Gadget by Changing its Behavior

This step explains how to change the behavior of an existing gadget. You will see how to extend the `ColoredTreeGadget` developed in Step 1 of this tutorial. You will change the behavior of the tree so that a contextual menu is displayed when the user clicks in the tree with the right mouse button. This contextual menu allows the user to change the color of the item where the user clicked, as shown in the following figure:



This step shows you how to perform the following tasks:

- *Choosing the Right Way to Modify the Behavior of an Existing Gadget*
- *Creating a Generic Interactor*
- *Creating a Dedicated Interactor*
- *Testing the New Interactor*

Choosing the Right Way to Modify the Behavior of an Existing Gadget

There are two ways to modify the behavior of an existing gadget:

1. Subclass the gadget and override its `handleEvent` method.

   This is the more logical solution when the behavior change is strongly linked with the kind of gadget it applies to.

2. Create a new interactor that will be set on the gadget.

   Creating a new interactor is a good solution when you want to temporarily modify the behavior of an object, or when the new interactor is generic, that is, you can set it on several different kinds of gadgets.

Creating a Generic Interactor

The second solution (creating a new interactor) has been chosen for this tutorial. The behavior to implement in this example (displaying a contextual menu after a right click) is generic—it can be used for any gadget class.

The complete code for the `ContextualMenuInteractor` class can be found in `ctxminter.h` and `ctxminter.cpp`. Here is the description of the class.

Purpose

The purpose of the `ContextualMenuInteractor` class is to provide a generic interactor that can handle a contextual menu, and that keeps the gadget behavior to which it is connected. For this reason, the `ContextualMenuInteractor` class should be a subclass of

the IlvGadgetInteractor class, that is, the predefined interactor class that is set on every gadget.

```
class ContextualMenuInteractor
: public IlvGadgetInteractor
{
public:
   virtual IlBoolean handleEvent(IlvGraphic* obj,
                                 IlvEvent&,
                                 const IlvTransformer* t);
   virtual IlBoolean shouldShowMenu(IlvGraphic*,
                                    IlvEvent&,
                                    const IlvTransformer*) const;
    virtual IlvPopupMenu* getMenu(IlvGraphic*,
                                  IlvEvent&,
                                  const IlvTransformer*) const = 0;
};
```

Each of these methods will now be seen in more detail.

The shouldShowMenu Method

The shouldShowMenu method should return IlvTrue if the interactor needs to display the pop-up menu on the specified event. The default behavior returns IlvTrue when the right button of the mouse is released. This method can be redefined in a subclass to display the pop-up menu on any other event.

```
IlBoolean
ContextualMenuInteractor::shouldShowMenu(IlvGraphic*,
                                         IlvEvent& event,
                                         const IlvTransformer*) const
{
    // The contextual menu is displayed by default when the right button
    // is released.
    return event.type() == IlvButtonUp && event.button() == IlvRightButton;
}
```

The getMenu Method

The getMenu method is pure, and therefore needs to be defined in a subclass to return the pop-up menu to display.

```
  virtual IlvPopupMenu* getMenu(IlvGraphic*,
                                IlvEvent&,
                                const IlvTransformer*) const = 0;
```

This method is called by the handleEvent method when the shouldShowMenu method has returned IlvTrue.

The handleEvent Method

Here is the implementation of the `handleEvent` method:

```
IlBoolean
ContextualMenuInteractor::handleEvent(IlvGraphic* obj,
                                      IlvEvent& event,
                                      const IlvTransformer* t)
{
    // Check that the object is a gadget.
    IlvGadget* gadget = accept(obj) ? IL_CAST(IlvGadget*, obj) : 0;
    if (gadget && gadget->isActive()) {
        // Is it time to display the contextual menu?
        if (shouldShowMenu(obj, event, t)) {
            // Get the menu.
            IlvPopupMenu* menu = getMenu(obj, event, t);
            // Show the menu.
            if (menu) {
                menu->get(IlvPoint(event.gx(), event.gy()),
                          /* transient */ 0);
                return IlvTrue;
            }
        }
    }
    // Default behavior of the gadget.
    return IlvGadgetInteractor::handleEvent(obj, event, t);
}
```

Creating a Dedicated Interactor

A subclass of the `ContextualMenuInteractor` class dedicated to the `ColoredTreeGadget` class will now be created. This subclass, called `ColoredTreeGadgetInteractor`, overrides the `getMenu` method to create a menu where some common colors are available, as shown in the following picture:

When the user selects a color, the color of the item where the mouse was located before the pop-up menu was displayed is changed.

Creating the Menu

Here is the implementation of the `getMenu` method:

```
IlvPopupMenu*
ColoredTreeGadgetInteractor::getMenu(IlvGraphic* graphic,
```

```
                                     IlvEvent& event,
                                     const IlvTransformer* t) const
{
    ColoredTreeGadget* tree = (ColoredTreeGadget*)graphic;
    // Find the item located at the event location.
    IlvTreeGadgetItem* item = tree->pointToItemLine(IlvPoint(event.x(),
                                                              event.y()),
                                                     (IlvTransformer*)t);
    if (!item)
        return 0;

    // Create the menu if needed.
    static IlvPopupMenu* menu = 0;
    if (!menu) {
        // Create the pop-up menu.
        menu = new IlvPopupMenu(tree->getDisplay());
        // Fill it with predefined colors.
        AddColor(menu, 0);
        AddColor(menu, "yellow");
        AddColor(menu, "green");
        AddColor(menu, "red");
        AddColor(menu, "blue");
        AddColor(menu, "gray");
        AddColor(menu, "brown");
    }
    // Set the item that asked for the contextual menu so that the menu
    // will be able to use it later.
    SetSelectedItem(menu, tree, item);
    // Return the menu.
    return menu;
}
```

After retrieving the item located under the mouse by using the `IlvTreeGadget::pointToItemLine` method, the menu is created with predefined colors by calling the static function `AddColor`.

Note: *The menu is created only once.*

Then the menu is connected to the tree and the tree item by calling the static member function `SetSelectedItem`. This static member function also sets the menu callback, as described in *The Menu Callback* on page 49.

Finally, the menu is returned.

The contents of the `AddColor` static function are shown in more detail:

```
static void
AddColor(IlvPopupMenu* menu, const char* color)
{
    // This static function is used to fill the contextual menu.
    IlvDisplay* display = menu->getDisplay();
    // A menu item is created with the specified color name.
    IlvMenuItem* item = new IlvMenuItem(color? color : "None");
    if (color) {
        // An IlvFilledRectangle object is created with the specfied color.
```

```
        IlvPalette* palette = display->getPalette(0, display->getColor(color));
        IlvFilledRectangle* rectangle =
            new IlvFilledRectangle(display,
                                   IlvRect(0, 0, 20, 20),
                                   palette);
        // The IlvFilledRectangle object is set as the picture of the item.
        item->setGraphic(rectangle);
    }
    // The item is inserted into the pop-up menu.
    menu->insertItem(item);
}
```

After creating the menu item by using the name given as the second parameter in the function, an `IlvFilledRectangle` is created with the right palette, and is set as the picture of the menu item. Then the menu item is inserted into the menu.

The Menu Callback

The details of how the color of an item is changed when a menu item is selected are now presented. The code of the menu callback is shown in detail:

```
static void
ChangeColor(IlvGraphic* g, IlvAny arg)
{
    IlvPopupMenu* menu = (IlvPopupMenu*)g;
    // Retrieve the selected item of the pop-up menu.
    IlvShort selected = menu->whichSelected();
    if (selected != -1) {
        // Retrieve its graphical representation.
        IlvSimpleGraphic* graphic =
            (IlvSimpleGraphic*)menu->getItem(selected)->getGraphic();
        // Use the second argument of the callback: A pointer to the colored
        // tree gadget object.
        ColoredTreeGadget* tree = (ColoredTreeGadget*)arg;
        // Retrieve the item whose color is going to be changed.
        IlvTreeGadgetItem* item =
            ColoredTreeGadgetInteractor::GetSelectedItem(menu);
        // Change the color of all the children of the parent item.
        if (item && item->getParent())
            tree->setChildrenBackground(item->getParent(),
                                        graphic
                                        ? graphic->getForeground()
                                        : (IlvColor*)0);
    }
}
```

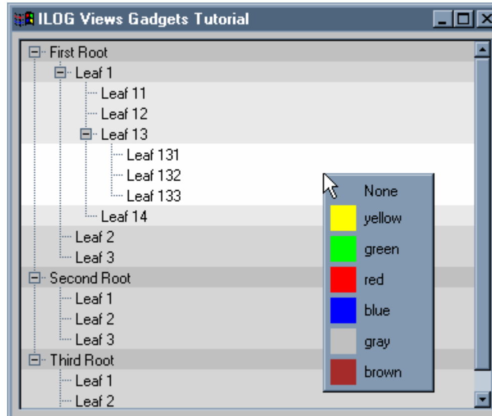
The color is taken from the picture of the selected menu item, and is set on the `ColoredTreeGadget` by calling the `setChildrenBackground` method.

Testing the New Interactor

The file `main.cpp` contains the code to test the `ColoredTreeGadgetInteractor` class. It is the same as the `main.cpp` file of Step 1, except that the interactor is set on the `ColoredTreeGadget` instance:

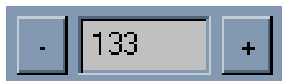
```
tree->setInteractor(new ColoredTreeGadgetInteractor());
```

This is the end of Step 2. The test program should look like this:



Step 3: Creating a Composite Gadget

This step explains how to create a new gadget, that is, a direct subclass of the `IlvGadget` class. The gadget to be created is a composite gadget made up of three gadgets: a text field and two buttons. The `UpDownField` gadget appears as follows:



This step shows you how to perform the following tasks:

- *Creating a Gadget Composed of Other Gadgets*
- *Handling Keyboard Focus in a Gadget*
- *Handling Events in a Gadget*
- *Adding Callbacks to a Gadget*
- *Testing the Composite Gadget*

Creating a Gadget Composed of Other Gadgets

Creating a composite gadget means creating a new gadget that is composed of one or more existing gadgets. This is the case for the `IlvTreeGadget`, which has internal

`IlvScrollBar` objects to allow scrolling operations. This is also the case for the `UpDownField` class created in this step.

Declaring the Composite Gadget

To keep pointers to the objects that `UpDownField` is composed of, protected member variables are used:

```
class UpDownField
: public IlvGadget
{
     ....
protected:
    IlvTextField* _textField;
    IlvButton*    _rightButton;
    IlvButton*    _leftButton;
   };
```

Creating the Components

First, the objects that make up the composite gadget must be created. The `UpDownField` class has a protected member function `init` that is called by each constructor of the class:

```
void
UpDownField::init(const char* label)
{
    // Compute the bounding boxes of each element.
    IlvRect r1, r2, r3;
    computeRects(r1, r2, r3);
    // Text field.
    _textField   =
        new IlvTextField(getDisplay(),
                         label,
                         r2,
                         getThickness(),
                         getPalette());
    _focusGadget = _textField;
    // Left Button.
    _leftButton  = new IlvButton(getDisplay(), "-", r1, getThickness(),
                                 getPalette());
    _leftButton->setCallback(_internal_Down, this);
    // Right Button.
    _rightButton = new IlvButton(getDisplay(), "+", r3, getThickness(),
                                 getPalette());
    _rightButton->setCallback(_internal_Up, this);
}
```

The `init` method first computes the bounding boxes of all the objects by calling the `computeRects` member function. Then, it creates and initializes the objects.

> ***Note:*** *The two buttons are given a callback. The left button callback will call the* `UpDownField::decrement` *method, and the right button callback will call the* `UpDownField::increment` *method.*

The Layout

Here is the description of the `computeRects` method responsible for the layout of the `UpDownField` class:

```
void
UpDownField::computeRects(IlvRect& r1,
                          IlvRect& r2,
                          IlvRect& r3,
                          const IlvTransformer* t) const
{
    IlvRect rect = _drawrect;
    if (t)
        t->apply(rect);
    r1.moveResize(rect.x(), rect.y(), (IlvDim)ButtonWidth, rect.h());
    IlvDim width = rect.w() - (2*(ButtonWidth + Margin));
    r2.moveResize(rect.x() + (IlvPos)(ButtonWidth + Margin),
                  rect.y(),
                  (IlvDim)IlvMax(width, (IlvDim)0),
                  rect.h());
    IlvPos deltaX = (IlvPos)(rect.w() - ButtonWidth);
    r3.moveResize(rect.x() + (IlvPos)IlvMax(deltaX, (IlvPos)0),
                  rect.y(),
                  ButtonWidth,
                  rect.h());
    r1.intersection(rect);
    r2.intersection(rect);
    r3.intersection(rect);
}
```

- `r1` is the bounding box of the left button.
- `r2` is the bounding box of the text field.
- `r3` is the bounding box of the right button.

This method is called each time the composite gadget is moved or resized. As with all graphic objects, the `applyTransform` method is called in both these cases:

```
void
UpDownField::applyTransform(const IlvTransformer* t)
{
    IlvGadget::applyTransform(t);
    IlvRect r1, r2, r3;
    computeRects(r1, r2, r3);
    _rightButton->moveResize(r3);
    _textField->moveResize(r2);
    _leftButton->moveResize(r1);
}
```

It is also used to query the composite gadget about its component bounding boxes.

Drawing the Components

The components of the composite gadget are now well located. The composite gadget draws them by calling the `draw` member function of each component:

```
void
UpDownField::draw(IlvPort* dst,
                  const IlvTransformer* t,
                  const IlvRegion* clip) const
{
    _textField->draw(dst, t, clip);
    _rightButton->draw(dst, t, clip);
    _leftButton->draw(dst, t, clip);
}
```

Redefining IlvGraphic Member Functions

Several member functions of the `IlvGraphic` class have been redefined to allow delegation to the components of the composite gadget. The `setPalette` method is given as an example:

```
void
UpDownField::setPalette(IlvPalette* palette)
{
    IlvGadget::setPalette(palette);
    _textField->setPalette(palette);
    _rightButton->setPalette(palette);
    _leftButton->setPalette(palette);
}
```

When the palette of `UpDownField` is changed, the same palette is set to each component of the `UpDownField`.

Finally, you need to override the `setHolder` method as follows:

```
void
UpDownField::setHolder(IlvGraphicHolder* holder)
{
   IlvGadget::setHolder(holder);
   _textField->setHolder(holder);
   _rightButton->setHolder(holder);
   _leftButton->setHolder(holder);
}
```

This method will be called each time `UpDownField` is added or removed from a holder. It ensures that all the components of `UpDownField` have the same holder. This is mandatory, as gadgets require a holder to behave properly.

Handling Keyboard Focus in a Gadget

The main difference between handling events in a graphic and in a gadget class is that a gadget can take the keyboard focus. Once a gadget has the keyboard focus, keyboard events are sent to this gadget. For more details, see “Focus Management” in Gadgets Main Properties.

Handling keyboard focus means:

- Reacting to focus events. When a gadget is about to be given the focus, it receives the IlvKeyboardFocusIn event. Similarly, when a gadget is about to lose the focus, it receives the IlvKeyboardFocusOut event.
- Handling the drawing of the focus. Two methods are involved in the drawing mechanism of the focus: IlvGraphic::computeFocusRegion and IlvGraphic::drawFocus. The first method gives information about the location of the focus drawing. The second method draws the focus.

It must be possible to give the focus to each component of UpDownField. When the focus is given to UpDownField, it is forwarded to the component that you chose to have it sent to. A protected member variable is added to keep a pointer to the current focus component of the UpDownField:

```
protected:
   IlvGadget*    _focusGadget;
```

A method named setFocus is also added to change the current focused object inside UpDownField:

```
void
UpDownField::setFocus(IlvGadget* gadget)
{
    IlvRegion region;
    // Send a focus_out event to the gadget that loses the focus.
    if (_focusGadget) {
        IlvEvent fo;
        fo._type = IlvKeyboardFocusOut;
        _focusGadget->computeFocusRegion(region, getTransformer());
        _focusGadget->handleEvent(fo);
        _focusGadget = 0;
    }
    _focusGadget = gadget;
    // Send a focus_in event to the gadget that receives the focus.
    if (_focusGadget) {
        IlvEvent fi;
        fi._type = IlvKeyboardFocusIn;
        _focusGadget->handleEvent(fi);
        _focusGadget->computeFocusRegion(region, getTransformer());
    }
    if (getHolder())
        getHolder()->reDraw(&region);
}
```

The method first sends an IlvKeyboardFocusOut event to the component that will lose the keyboard focus. Then it sends an IlvKeyboardFocusIn event to the new focused component. Finally, the modified region is redrawn using the IlvGraphicHolder API.

The following shows how the focus is drawn. The computeFocusRegion and drawFocus methods are involved in this process:

```
void
UpDownField::drawFocus(IlvPort* dst,
                       const IlvPalette* palette,
```

```
                       const IlvTransformer* t,
                       const IlvRegion* clip) const
{
    _focusGadget->drawFocus(dst, palette, t, clip);
}

void
UpDownField::computeFocusRegion(IlvRegion& region,
                                const IlvTransformer* t) const
{
    _focusGadget->computeFocusRegion(region, t);
}
```

The `UpDownField` delegates to the current focused object. The next section shows how to change the keyboard focus of the `UpDownField`.

Handling Events in a Gadget

The entry point for events in a gadget class is the `IlvGadget::handleEvent` method. This method is called by the gadget holder when it receives an event that should be handled by the gadget. The `UpDownField::handleEvent` is very simple, as it delegates most of the events to the focused object (pointed by the `_focusGadget` member variable):

```
IlvBoolean
UpDownField::handleEvent(IlvEvent& event)
{
    IlvBoolean result = IlvFalse;
    switch (event.type()) {
    case IlvButtonDown:
        {
            // Changing focus on click
            IlvRect r1, r2, r3;
            IlvPoint evp(event.x(), event.y());
            computeRects(r1, r2, r3,  getTransformer());
            if (r2.contains(evp) && _focusGadget != _textField)
                setFocus(_textField);
            else
            if (r3.contains(evp) && _focusGadget != _rightButton)
                setFocus(_rightButton);
            else
            if (r1.contains(evp) && _focusGadget != _leftButton)
                setFocus(_leftButton);
            result = _focusGadget->handleEvent(event);
            break;
      }
    case IlvKeyDown:
        {
            // Moving focus with the Tab key.
            if (event.data() == IlvTab &&
                (!(event.modifiers() & IlvShiftModifier)) &&
                (!(event.modifiers() & IlvCtrlModifier))) {
                if (_focusGadget == _textField)
                    setFocus(_rightButton);
                else
                if (_focusGadget == _rightButton)
                    setFocus(_leftButton);
```

```
                else
                if (_focusGadget == _leftButton)
                    setFocus(_textField);
                return IlvTrue;
            }
            // Moving focus with the Shift Tab key.
            if (event.data() == IlvTab &&
                ((event.modifiers() & IlvShiftModifier)) &&
                (!(event.modifiers() & IlvCtrlModifier))) {
                if (_focusGadget == _leftButton)
                    setFocus(_rightButton);
                else
                if (_focusGadget == _rightButton)
                    setFocus(_textField);
                else
                if (_focusGadget == _textField)
                    setFocus(_leftButton);
                return IlvTrue;
            }
        }
    default:
        result = _focusGadget->handleEvent(event);
    }
    return result;
}
```

The main part of the `handleEvent` method deals with keyboard focus management. The `IlvButtonDown` case is responsible for changing the current focused object when a button down event is received by the composite gadget. The `IlvKeyDown` case is responsible for changing the current focus object when a TAB key or Shift-TAB key event is received by the composite gadget. The default case simply delegates to the current focused object.

Adding Callbacks to a Gadget

As shown at the beginning of this step, the left and right buttons of `UpDownField` are assigned a callback to allow user notification when pressed. Two callbacks are now defined: a `Down` callback, which will be called each time the left button is pressed, and an `Up` callback, which will be called each time the right button is pressed.

These callbacks are declared in the `getCallbackType` method:

```
IlvUInt
UpDownField::getCallbackTypes(const char* const**      names,
                              const IlvSymbol* const** types) const
{
    IlvUInt count = IlvGadget::getCallbackTypes(names, types);
    AddToCallbackTypeList(count, names, types,
                          "Down", downCallbackType());
    AddToCallbackTypeList(count, names, types,
                          "Up", upCallbackType());
    return count;
}
```

> ***Note:** Redefining this method is not mandatory, but will allow easier integration of the* `UpDownField` *in an editor (such as IBM ILOG Views Studio), since it will be possible for the editor to query the list of callbacks handled by the* `UpDownField`*.*

Then, the `UpDownField::increment` method is implemented to call the `Up` callback and the `UpDownField::decrement` method is implemented to call the `Down` callback:

```
void
UpDownField::increment()
{
    callCallbacks(upCallbackType());
}

void
UpDownField::decrement()
{
    callCallbacks(downCallbackType());
}
```

Testing the Composite Gadget

The file `main.cpp` contains the code to test the `UpDownField` class. It shows how to implement the `Up` and `Down` callbacks of the `UpDownField` class to change the value of the text field—the `Down` callback will decrement the text field value, while the `Up` callback will increment it.

First, a container is created and an instance of the `UpDownField` class is added to it.

```
IlvDialog* dialog = new IlvDialog(display,
                                  title,
                                  title,
                                  IlvRect(0, 0, 100, 100));
UpDownField* but = new UpDownField(display, IlvRect(5, 5, 100, 23) , "0");
dialog->addObject(but);
```

Then, the callbacks of the `UpDownField` are set:

```
but->setUpCallback(Increment);
but->setDownCallback(Decrement);
```

Here is the implementation of the callbacks:

```
static void Increment(IlvGraphic* g, IlvAny)
{
    char buffer[1000];
    UpDownField * obj = (UpDownField*)g;
    const char* label = obj->getLabel();
    if (label && *label) {
        IlvInt value = ((IlvInt)atof(label))+1;
        sprintf(buffer, "%ld", value);
        obj->setLabel(buffer, IlvTrue);
    } else
```

```
        obj->setLabel("0", IlvTrue);
}

static void Decrement(IlvGraphic* g, IlvAny)
{
    char buffer[1000];
    UpDownField * obj = (UpDownField*)g;
    const char* label = obj->getLabel();
    if (label && *label) {
        IlvInt value = ((IlvInt)atof(label))-1;
        sprintf(buffer, "%ld", value);
        obj->setLabel(buffer, IlvTrue);
    } else
        obj->setLabel("0", IlvTrue);
}
```

These callbacks retrieve the text field value of the `UpDownField` instance by calling `UpDownField::getLabel`, modify it by adding or removing the value 1, and set it as the label of the text field by calling `UpDownField::setLabel`.

This is the end of Step 3.

Index

C

C++
  prerequisites **3**
customizing gadgets tutorial **35**

M

manual
  naming conventions **4**
  notation **3**

N

naming conventions **4**
notation **3**

T

tutorials
  customizing gadgets **35**
  viewfile **3**

V

viewfile tutorial **3**